プレッシャーズ特別企画

第3章 無駄のないデバイス・ドライバを設計する

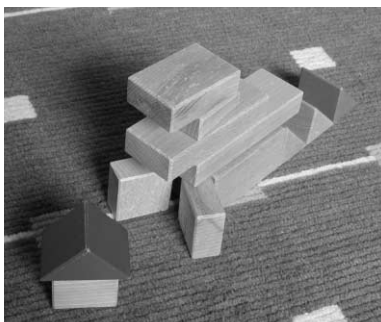
地道な分析と自由な発想を両立しよう

森 孝夫

ここでは，デバイス・ドライバの設計について解説する．デバイス・ドライバは，ハードウェア（デバイス）と密接に関連しているので，これを作成するにはソフトウェアからデバイスを操作するための知識や，割り込みに関する知識のほか，移植しやすいようにモジュール化する技術も必要となる．　（編集部）

デバイス・ドライバとは，特定のハードウェア（デバイス）を制御するために用意されるソフトウェア・モジュールです．デバイス・ドライバが用意されていると，アプリケーション・ソフトウェアはデバイスの機能を簡単に利用することができます．

組み込みシステム開発では，製品に新機能を追加するときなどに新しいデバイスを搭載することがあり，その際にデバイス・ドライバの設計が必要となります．本稿では，移植性に優れ，なおかつデバイスの性能を引き出すことができるデバイス・ドライバを設計するノウハウを解説します．

写真1
積み木
積み木を見ると積んでみたくなる．積んでいるうちに，新しい形を作り出してしまう．積み木やブロックは，そのような不思議な性質を持っているように思われる．

1 機能美のあるシステムを目指して

デバイス・ドライバに限らず，組み込みソフトウェアは「無駄がない」ということを強く求められます．なぜなら組み込みシステムは，ぎりぎりの性能，ぎりぎりまで小さくしたハードウェアで構成されることが多いからです．

組み込みシステム開発では，ユーザ（顧客）に「良いものを安く」提供するために，製品に必要な機能を満たせるハードウェアの中で，できるだけ安価なものを使おうとします．その結果，多くの場合，性能面ではぎりぎりのハードウェアが使われることになるのです．

組み込みシステムに対する要求は，機能のほかにも，リアルタイム性や低消費電力，（取り付けスペースなどの関係で）回路を小さくしてほしいという要求など，次々と出てきます．これらを達成するためには，いずれも無駄をなくしたシステム設計が必要です．無駄をなくした設計こそが，結果として，ユーザに価値を提供することになります．

この，無駄のないシステム作りを支えているのが，組み込みソフトウェアです．組み込みソフトウェアがシステムの性能を極限まで引き出すからこそ，ぎりぎりの性能や大きさのシステムに価値ある機能が搭載できるのです．そのスマートさはまさに「機能美」と呼べるものではないかと思います（コラム「組み込みシステムには機能美がある」を参照）．

● 組み込みソフトウェアの組み立て方

皆さんは，小さなころに「レゴ」で遊んだことはありませ

KeyWord　デバイス・ドライバ，機能美，設計，大喜利，あいうえお作文，Sound Blaster，隠れた要求，責務

COLUMN
組み込みシステムには機能美がある

筆者は，電化製品や自動車，ロケットの模型などに触れたとき，感覚的に「機能美」を感じることがあります．「知性の美しさ」といった方が，筆者の感覚を正しく表しているかもしれません．でき上がった物の形や動きを見るだけで，「機能を実現するためのスマートな知性」を感じ，そこに美しさを感じることがあるのです．音楽，美術，建築物，はたまたスポーツのプレー，時には動物の本能的な行動を見ても，同じような感覚を抱くことがあります（**写真A**）[注A]．

そのような「知性」を感じる製品について設計思想を聞いてみると，機能を実現するためのきめ細かい工夫や，無駄をそぎ落とすための工夫，信頼性を保つための工夫が随所に施されています．

製品を見ただけでは，製品の設計思想を知る術はありません．なのに，見ただけで感覚的に，しかも一瞬にして知性を感じるというのは不思議なものです．この感覚が生まれる理由を筆者は合理的に説明できないのですが，エンジニアを続けていると，こうした機能美を感じる感性みたいなものが備わってくるような気がします．

仕事で大切なのは，「良い仕事をした」と思えるような，そして周りにも「良い仕事をした」と思ってもらえるような仕事をすることです．われわれ組み込みシステムのエンジニアは，ユーザに安くて良い物を提供しようと追求することで，自然と「機能美」を作っているように思います．それは，エンジニアにとっての大きな楽しみなのではないでしょうか．

写真A 「機能美」を感じる製品の例
筆者の愛車．最近入手したばかり．

注A：実は逆もある．ちょっと触って「これはバグが出る気がする」と感じることがあるのだ．経験的に言って，エンジニアが持つ「なんか怪しい…」という感覚はかなり当たる．

んか？ 人は誰しも，レゴ・ブロックを見るとなぜか，とりあえず“カポッ”と組み合わせてみます．そして，いろいろ組み合わせて試行錯誤を繰り返し，そのうち新しい形を作り出します．これは，レゴ・ブロックや積み木遊びが持つ不思議な性質であるように思われます（**写真1**）．

人間には，レゴ・ブロックや積み木のような「材料」が転がっているのを見ると，それを組み合わせたり，分類したりして，自身がしっくりくるような「新しい設計」を考える性質があるような気がします．

組み込みソフトウェアの設計も，まずは「材料」探しから始めてみましょう．組み込みソフトウェアの材料とは，「組み込みソフトウェアが内部で行うこと」です．「内部で行うこと」をまとめてプログラミングすると，ソフトウェアのモジュールになります．そして，モジュールの作り方と組み合わせ方が「設計」というわけです（p.117のコラム「設計とは大喜利である」を参照）．

「内部で行うこと」を決めるためには，組み込みシステムで実現したいと思っている機能のほか，提供されたハードウェアやOS，プログラミングに使用するライブラリなどの仕様をよく知る必要があります．例えば，「前進する」という機能をマイコンとモータで実現しようと思ったら，「モータに電圧をかける」ことが必要です．そのためには，マイコンのI/Oポートとモータの接続を知る必要があり，I/Oポートに電圧をかけるための命令も知る必要があります（**図1**）．そして，知った内容をうまく組み合わせて，モジュール構造を作るのが「設計」です．

組み込みソフトウェアの設計とは，要求された機能や性能を，与えられたシステムの制約下で実現する方法を考える行為です．その第一歩は，要求仕様書，ハードウェアの仕様書，OSの仕様書などをよく読み，要求と制約の両方を把握することです．「機能美」は，こうした地道でまっとうな努力の積み重ねから生まれるのです．

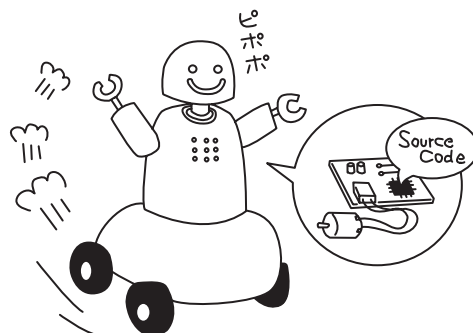



図1 要求と制約を把握して機能や構造を考える
ロボットを前進させるためには，マイコンとモータの接続や，I/Oポートへの命令の仕方を知る必要がある．これらの「知った内容」をうまく組み合わせてモジュール構造を作るのが「設計」である．

● デバイス・ドライバは機能を提供する

若い皆さんはご存じないかもしれませんが，昔は家庭や学校に「石油ストーブ」というものがありました．石油ストーブは，灯油を入れて火をつけると，部屋を暖めてくれるというものでした（**図2**）．石油ストーブは，いろいろな目的に使われたものです．部屋を暖めるという本来の目的のほか，パンを焼くのに使われたり，学校ではぞうきんを乾かすのにも使われました．

しかし今考えてみると，石油ストーブが燃焼する「仕組み」はよく知りませんでした．燃焼のさせ方だけは知っているのですが，中でどう燃焼しているのかは全然知らなかったのです．それでも道具としてきちんと使いこなせていました．つまり，「暖める」，「焼く」，「乾かす」という機能を実現するためには，「燃焼する」という機能を利用すればよいわけで，燃焼する方法まで知る必要はなかったのです．

アプリケーション・ソフトウェアとデバイス・ドライバの関係も，このようなドライ（？）な関係にする必要があります．つまり，アプリケーション・ソフトウェアに対して，デバイス・ドライバを以下のような存在とします（**図3**）．

- 必要な機能を提供する
- 利用しやすいインターフェースを提供する
- デバイスの詳細を隠ぺいする

こうすることで，アプリケーションはデバイスの詳細を知ることなしに，機能だけを簡単に利用できるのです．

● デバイス・ドライバは水面下で頑張る

白鳥というのは，水上からは優雅に泳いでいるように見えても，水面下ではバタバタと水をかいているのだ，という話をよく聞きます．これは，デバイス・ドライバにも言えることです．例えば，DVDレコーダなどでユーザが録画を開始すると，「水面下」では大変な速度でデバイス・ドライバとデバイスが協調動作し，機能を実現しているのです．そのような「水かき」の様子を，1990年代にパソコン上の音源として流行したハードウェア「Sound Blaster」を例に解説します(2)注1．

Sound Blasterは，PCM（palse code modulation）音源の録音，再生を行う，パソコン用のハードウェアです（**写真2**）．このハードウェアは，パソコンのメイン・ボードの拡張コネクタに差し込むボードや，PCカードという形で販売されていました．それらをパソコンに接続し，デバイス・ドライバをインストールすると，家庭のパソコンがサウンド・システムに早変わりするのです．当時はまだ，パソコンから音が出るのが当たり前ではなかったので，楽しいハードウェアでした．このサウンド・システムの主な機能である，録音機能の構造を**図4**に示します．

Sound Blasterは，音楽CDと同じサンプリング周波数44.1kHzの波形を扱うことができました．44.1kHzのデータ処理，つまり，1秒間に44,100回のデータ処理を行うということは，処理の周期は約20 μsです．扱う対象が音声であることを考えると，大きな誤差は許されません（p.118のコラム「時間に関する要求を忘れずに」を参照）．しかし，当時のパソコンの性能では，ソフトウェアだけでこの周期で確実に処理を行うのは不可能に近い時代でした．そこで，Sound Blasterハードウェア上に「正確な周期で処理できるデバイス」を搭載し，高い周期精度が求められる個所はデバイスにやらせてしまおうと考えたわけです．その結果，このシステムは，ハードウェアとソフトウェアが絶妙に協調動作する，なかなか面白いシステムになっています．

注1：Sound Blasterは現在も継続して開発・販売されているが，本稿では1990年代当時のものについて言及している．

図2　仕組みは知らなくても，使い方だけ知っていれば使える

図3　難しいことはデバイス・ドライバにお任せ

COLUMN
設計とは大喜利である

皆さんは，「あいうえお作文」というのをご存じでしょうか．これは，文章の頭文字だけを決めておいて，後はその場の状況に合わせて「うまいこと文章を作る」という，一種の知的パズルです．

ところが，この「うまい」というのが大変難しいのです．文章のテーマが場に合っていなければならないうえ，何かと「掛かって」いなければならず，最後にはオチが必要です．ほかにもいろいろ隠れた「要求」が存在します．その点は，組み込みシステム設計や組み込みソフトウェア設計と何だか似ているような気がします．

● 大喜利も設計もtry & fail

あいうえお作文のように，何かお題を決めて「うまい解を出す」行為のことを「大喜利」といいます．大喜利と設計は，いろんな点でよく似ています．

まず，正解は複数存在する可能性があります．「うまい」となるような作文も複数存在するでしょうし，要求を満たせるような設計解も，複数存在しうるのです(**図A**)[注B]．

思考過程も似ています．あいうえお作文も設計も，思考過程においてはtry & failを繰り返します．つまり，「しっくりくる」までいろいろな解の候補を出してみて，「これだ」と思ったものを解とするわけです．そして，「しっくりくる」瞬間は，茂木 健一郎氏が言うところの「アハ体験」[(1)]に近いような満足感を得ることができます．

このように考えると，組み込みシステムやソフトウェアの設計の難しさと楽しさの秘密が分かってくるのではないでしょうか．ソフトウェア設計というのは自由度が高い上に，少し考えただけで「しっくりくる」ような設計が得られるとは限らないのです．加えて，「しっくりくる」とか「うまい」という基準が定義しづらいという難しさもあります．でも，難しいからこそ，うまく設計できたときには大きな満足感が得られるのです．それに，完成すると「物が動く」という楽しさもあります．

注B：設計や大喜利のような問題のことを「数理的逆問題」と呼ぶ．数理的逆問題の場合，目指す「結果(要求を満たしたシステム，場内の大爆笑)」を得られるように，「原因」となる「設計解」や「ネタ」を考える．

● 頭の柔軟体操を欠かさずに

いかにソフトウェア工学が成熟したとしても，ソフトウェアの設計は「誰でも簡単にできる」ようにはならないと思います．それは，ある一定以上の自由度の高さは残る気がするからです．やはり，

- 自分で設計解の候補を考え出す力
- 「しっくりくる」まで繰り返し考える粘り強さ
- 求められている(隠れた)要求を把握しようとする意識

など，エンジニア自身の思考力と精神構造が必要とされるように思います．

筆者から若い皆さんに提案します．組み込みシステム開発やソフトウェア設計に携わるときは，

- まず，自由にいろいろと考えてみましょう．
- 設計解の候補をたくさん考えてみるようにしましょう．
- 設計解が要求を満たすか確認するようにしましょう．
- いくつか考えた設計解の候補が，それぞれ誰に対してどんなメリットを生み出すのかを考え，比較してみるようにしましょう．

これらを日々行って習慣づけていくと，設計に求められる隠れた要求を見いだす「要求把握能力」，要求から解くべき問題を設定する「問題発見能力」などを，自然と身に付けることができます．そうすると「しっくりくる」感覚に自信が持てるようになり，満足感も上がってくることでしょう．

ところで最近，ある大学で，とんでもない大喜利の達人に出会いました．その先生にあるお題を出したところ，3秒くらいであいうえお作文を1個，10分くらいで3個も作り出してしまったのです(う〜む，すごい)．これからは筆者も，設計能力を磨くために，毎週「笑点」か「エンタの神様」を見ようと思います．

図A 「うまい」あいうえお作文の例

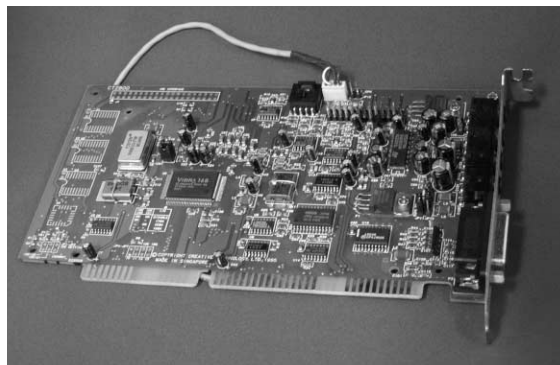



写真2 Sound Blasterを搭載したサウンド・ボード
1990年代の製品．「Sound Blaster VIBRA 16S」を搭載している．

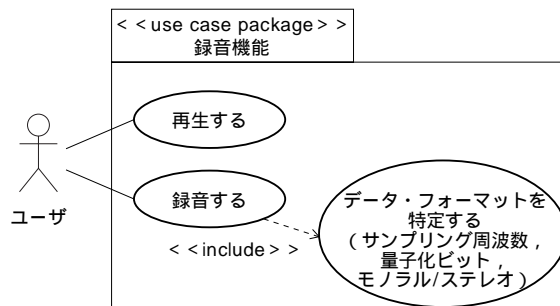



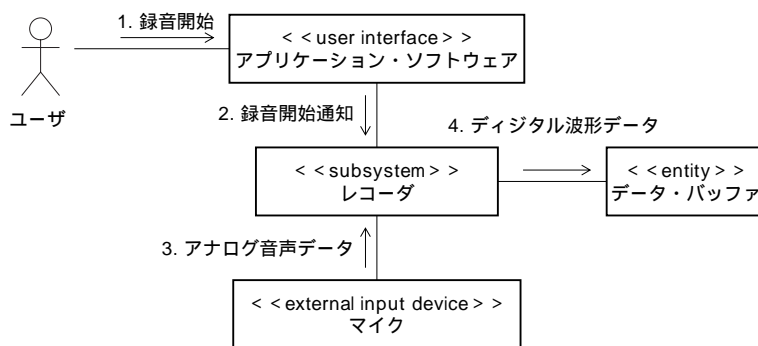
図4 サウンド・システムの録音機能
UMLのユース・ケース図で表現してみた．「録音する」とは「データ・フォーマットを特定する」ことを含む．

3

Sound Blasterの録音のユース・ケース(機能)に関する内部動作を**図5**に示します．デバイス・ドライバを利用するアプリケーション・ソフトウェアは「録音開始」をデバイス・ドライバに通知するだけで，後は何もしません．しかし，その通知を受けたデバイス・ドライバとデバイス(**図5**中の「レコーダ」)は，アプリケーション・ソフトウェアの知らないところで，約20 μsごとに動作を繰り返すのです．まさに「水面下の白鳥」ですね．

サブシステム「レコーダ」を詳細化したモデルで，原理を説明しておきましょう(**図6**)．図中の「サンプラ」は，「マイクからの音声データを，正確な周期でサンプリングすることができる部品」です．図中で太線になっている四角は，並行処理可能なプロセスを示しています．録音が開始されると「サンプラ」は，内部バッファにどんどんデータをため込んでいきます．半分までたまると，「ストア」モジュールに通知します．通知を受けた「ストア」モジュールは，内部バッファにたまったデータをデータ・バッファの領域にコピーします．その間，「サンプラ」は内部バッファの残り半分の領域にデータをためていきます．

「サンプラ」が行うサンプリングの実現手段として使用

図5
サウンド・システムの録音の動作
UMLのコラボレーション図で表現してみた．「レコーダ」は録音開始通知を受けて，マイクで拾った音声をディジタル・データとして蓄積していく．

COLUMN 時間に関する要求を忘れずに

皆さんは家で鉢植えを育てたことはありますか．家に植物があると，それだけで和みますよね．しかし，ほったらかしではいけません．元気に育てるためには，一定の間隔で水や肥料をやる必要があります．

このような「一定の間隔で○○する必要がある」というのは，時間に関する要求です．「1日1回，水をやってください」とか「毎月1回振り込んでください」には，時間に関する要求が含まれています．

「この期限までに○○する必要がある」というのも，時間に関する要求です．例えば，「この仕事を明日までに完了してください」には，時間に関する要求が含まれています．

本稿で取り上げたデバイス「Sound Blaster」で言えば，サンプリング機能がA-D(analog to digital)変換する周期についてはかなり厳しい要求が課せられます．44.1kHzで動作する場合には，「ほぼぴったり1/44100sの周期で，アナログ・データをディジタル・データに変換して，メモリにため込まなければならない」のです．これが狂えば，聞いていて違和感のあるデータが録音されてしまいます．

● 隠れた時間要求を意識する

ここまで挙げたものは，時間通りに行うことが「必須」とされるものです．時間に関する要求の中でも，必須とされる要求に関しては，人が見逃すことはほとんどありません．ところが，時間に関する要求の中には，見逃しやすいものや定義しづらいものもあるのです．

例えば，ボクシングでは，相手から目を離してはいけません．1～2秒目を離したら，その時間内にパンチを受けてしまいます．しかし，まばたきなどで一瞬は相手から目が離れることもあるはずです．そこには，例えば「0.**秒以上目を離してはいけない」というような，隠れた時間要求が存在するように思います．

組み込みソフトウェアで言えば，メイン・ループ内で毎回行われている処理には，隠れた時間要求があることが多いようです．「だいたい10ms～30msに1回，その処理が行われればよい」，「割り込みなどの処理がたまって，時にはメインの1周の時間が多少長くなっても構わない」，「しかし，最悪でも100msに1回は処理したい」というような要求が潜んでいるのです．つまり，「定義しづらいのだが，確かに時間に関する要求は存在する」のです．

このような要求を明文化するかどうかはケース・バイ・ケースです．けれども，このような要求が存在していることを意識するかしないかでは，ソフトウェアのでき上がりが違ってきます．たとえ明文化されていなくても，時間に関する要求を忘れずに考察するようにしましょう．

されたのが，Sound Blasterデバイスと DMA（direct memory access）コントローラの組み合わせです．DMAコントローラはバス・マスタ・デバイス（CPUがアクセスできるメモリ領域に，自ら直接アクセスできるタイプのデバイス）です．そのため，Sound BlasterとDMAコントローラを組み合わせることによって，内部バッファに直接録音データを書き込んでいました（**図7**）．

ここで，「内部バッファなどを介さずに，データ・バッファの領域に直接書き込めばよいのではないか」と思われるかもしれませんが，当時のパソコンのハードウェアの制約下では，そうもいかなかったのです．内部バッファがわざわざ用いられていたのは，当時のパソコンのDMAコントローラが最大64Kバイトのメモリしか扱えなかったからです．64Kバイトというと，16ビットのステレオ・データでは16,000サンプル，つまり0.36秒しか記憶できません．これでは音楽にならないので，まずは小さな内部バッファにためておき，半分までたまるたびにデータ・バッファに移し変えるという構造にしていたのです．そして，内部バッファが半分たまったことを通知する手段が，割り込みでした．割り込みにより「ストア」モジュールが起動し，内部バッファからデータ・バッファへのデータ・コピーが行われました．

こうして見ていくと，**図6**はソフトウェアとハードウェアの協調設計のはしりとも言える図になっていることが分かります．当時のハードウェア制約がこのようなユニークな設計を生んだかと思うと，とても興味深いです．この「水面下の水かき」は，なかなかの「機能美」だと思いませんか．

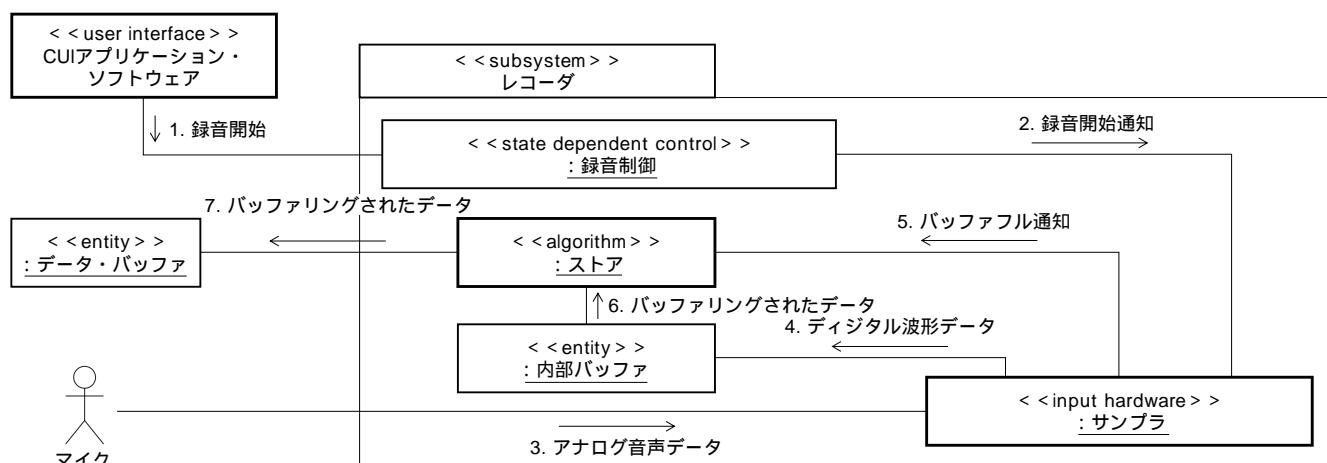



図6　サブシステム「レコーダ」の詳細

図5中の「レコーダ」を詳細化したもの．ソフトウェアとハードウェアがここまで強く結合している場合は，再利用やドメイン分割というより，ハードウェア中心のこのモデルの方が分かりやすい．また，「このシステムは何か（What）」を示すのに有効なモデルでもある．

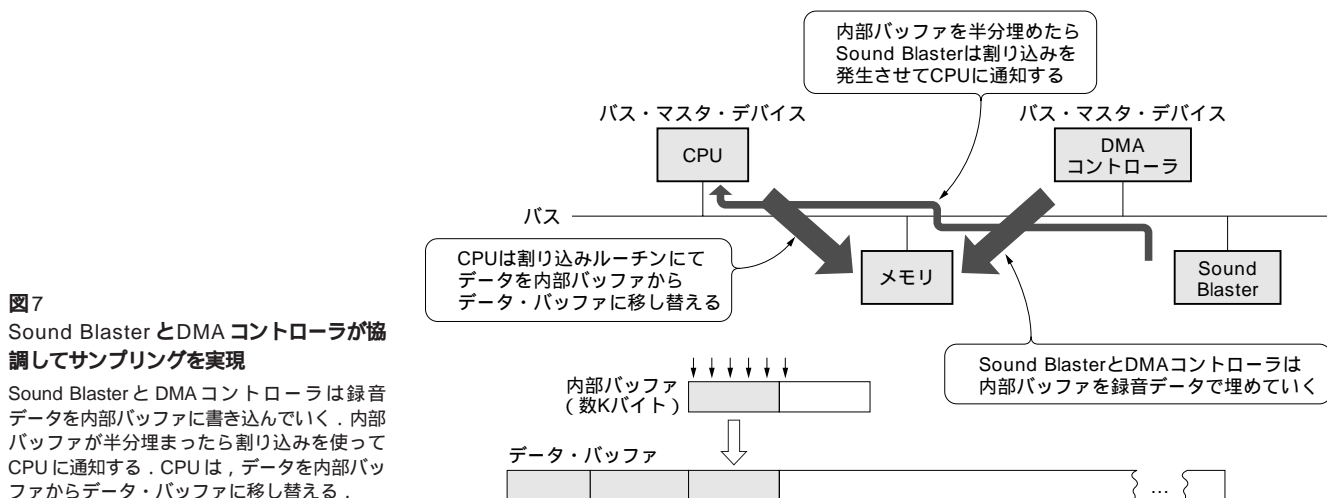



図7
Sound BlasterとDMAコントローラが協調してサンプリングを実現

Sound BlasterとDMAコントローラは録音データを内部バッファに書き込んでいく．内部バッファが半分埋まったら割り込みを使ってCPUに通知する．CPUは，データを内部バッファからデータ・バッファに移し替える．

2 デバイス・ドライバを設計する

ここまで，Sound Blasterというデバイスを利用したシステムの姿を概観してきました．今度は，このシステムで使えるデバイス・ドライバを設計してみましょう．

● デバイス・ドライバの責務を決める

まず，システム内の構成要素の責務(「何をするのか」という役割)を分析し，Sound Blasterのデバイス・ドライバの責務を決めます．**図6**を見ながら考えてみましょう．図中の「サンプラ」はハードウェア(部品)です．ハードウェアの責務は，ハードウェアの仕様で決まります．「サンプラ」に相当する部品の仕様を確認すると，前述の通り，Sound BlasterとDMAコントローラの連携により，アナログ波形をディジタル値に変換し，内部バッファに入れるところまでをやってくれています[注2]．

データ・バッファは，一時的にデータを蓄積する中間バッファではなく，最終的に音を記録する領域です．最終目的のための領域なので，これはアプリケーション・ソフトウェアから指定した方が，使い勝手が良いでしょう．よって，データ・バッファはアプリケーション・ソフトウェアの管理範囲(すなわち，デバイス・ドライバの責務ではない)とします．

録音中，停止中などといった状態を管理する機能は，移植性を考えて，デバイス・ドライバの責務としておきましょう．こうすれば，アプリケーション・ソフトウェアを新たに作るたびに状態管理を作る必要はなくなります．

そうすると，デバイス・ドライバの責務は以下のように定義できます．

- 録音中や停止中などの状態を管理する．
- 録音中は，「サンプラ」から内部バッファに送られてくるディジタル波形データを，アプリケーション・ソフトウェアから指定されたデータ・バッファにコピーする．

これにより，デバイス・ドライバの範囲は，**図6**のブロックのうち，「録音制御」，「ストア」の二つのモジュールと，内部バッファ領域に決まります．

このように，デバイス・ドライバの責務を決める行為は，システム内の構成要素の「責務分担」と密接な関係があります．デバイスだけの視点からデバイス・ドライバの責務をとらえず，システムの視点からもデバイス・ドライバの責務を割り振るようにしましょう．

注2：詳しくは，参考文献(2)のChapter 2やChapter 3を参照のこと．

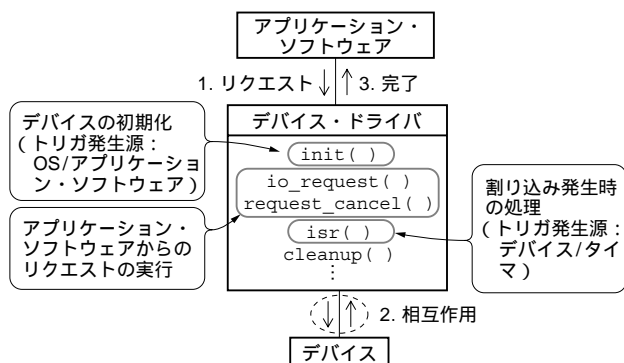



図8　一般的なデバイス・ドライバのインターフェース

一般的なデバイス・ドライバのインターフェースとしては，デバイスの初期化/終了処理，アプリケーション・ソフトウェアからのリクエスト受け付け処理，デバイスやタイマからの割り込み処理などが挙げられる．

● デバイス・ドライバのインターフェースを設計する

デバイス・ドライバの責務を設計できたら，今度はデバイス・ドライバのインターフェースを設計します．デバイス・ドライバのインターフェースは，大きく次の4種類に分けられます(**図8**)．

- デバイスの初期化/終了
- アプリケーション・ソフトウェアからの通知を受け取る
- デバイスからの通知を受け取る
- 周期処理を行う

1) デバイスの初期化/終了

デバイスを扱う場合，初期化は必ずと言ってよいほど必要になります．また，WindowsやLinuxのような高級OSの場合，OSの初期化時のほか，デバイス・ドライバの新規インストール時にも初期化処理が行われます．通常の組み込みシステムの場合には，システムの初期化時にデバイスの初期化も行われます．初期化処理の中身は，ほとんどの場合，デバイスのリセットやコンフィグレーションになります．Sound Blasterの場合には，Sound BlasterそのものとDMAコントローラの初期化を行います．

終了は，デバイスを使用しなくなったときや，電源OFF時に行う処理で，必要があれば記述します．

2) アプリケーションからの通知を受け取る

アプリケーション・ソフトウェアからの通知を受け取るインターフェースは，上位(アプリケーション・ソフトウェア)にとって必要な機能を簡単に呼び出せるように設計し

ます．設計の際は情報隠ぺいを意識しましょう．そのためには「アプリケーション・ソフトウェアの目的」を意識し，目的と関係のない情報は隠ぺいするようにします．

例えば，Sound Blasterであれば，アプリケーション・ソフトウェアの目的は，サンプリング周波数やモノラル/ステレオを設定して録音することです．これらを簡単に指定できるようにします．そして，目的とは関係のない情報，例えばSound BlasterがCPUのどのI/Oポートに接続されているかは，モジュール内部に隠ぺいします．

3）デバイスからの通知を受け取る

デバイスからの通知を受け取る仕組みとしては，割り込みが用いられます．従って，デバイスから通知を受け取るためのインターフェースは，割り込みルーチンから呼び出されます．

4）周期処理を行う

周期的に処理を行うためのインターフェースは，タイマ割り込みやOSの周期起動タスクから呼び出されます．

いずれにしても，デバイス・ドライバのインターフェースは，割り込みやタスクそのものにはせず，あくまで，割り込みやタスクから「呼び出される」ように設計します．そのようにすることでプラットホームとデバイス・ドライバを分離でき，デバイス・ドライバの再利用性が向上します．

Sound Blasterのデバイス・ドライバのインターフェース設計例を，**表1**に示します．

● デバイスの詳細を分析する

デバイス・ドライバのインターフェースを設計できたら，次はデバイスの詳細を分析し，「内部で行うこと」を詳細まで明らかにしていきます．ここで，ハードウェアやOS，マイコンのマニュアルを読むことになります．地道ですが，大切な作業です．

Sound Blasterのハードウェア・マニュアルを調べると，「DSP命令（Digital Sound Processor Command）」を使ってさまざまな設定ができることが分かります（**表2**）．DSP命令の中には，時定数を設定するコマンドや，内部バッファの大きさを指定するコマンド，録音の開始/停止を行うコマンドなどが定義されています．また，Sound BlasterのDSP命令を送るためには，どのアドレスにアクセスすればよいかも確認しておきましょう（**表3**）．

● デバイス・ドライバの構造を設計する

デバイス・ドライバの「内部で行うこと」が分かってきたら，今度はデバイス・ドライバの構造を設計してみましょう．構造とは，ソフトウェアのモジュール同士の結びつき方のことです．

構造を設計するためには，材料である「内部で行うこと」を並べて，それらの「目的と手段の関係」を整理します．例えば，「周波数を設定する」と「時定数コマンドを送る」という二つの行為があったとすると，前者が目的で，後者が手段です．「録音を開始する」と「コマンド2Chを送る」も，前者が目的で，後者が手段です．

機能間の「目的と手段」の関係を整理していくと，「機能の階層構造」を構築できます．ソフトウェアでは多く場合，目的側に位置するモジュールが，手段側に位置するモジュールを呼び出します．従って，機能の階層構造が構築できれば，おのずとソフトウェアの構造も決めることができるのです．

図9に，Sound Blasterデバイス・ドライバの構造を示

表1　Sound Blasterデバイス・ドライバのインターフェース設計例

インターフェース	関数名	処理内容
初期化	（割り込みベクタの宣言など）	割り込み使用の宣言
上位からのリクエスト	`void init_device( void )`	録音ハードウェアの初期化
	`void config_property( uint frequency, uint sample_size, uint mono)`	周波数、量子化幅、モノラル/ステレオの種別を録音ハードウェアに設定
	`void set_databuf( void *buf )`	録音用データ・バッファを設定
	`int start_rec( void )`	録音開始
	`int stop_rec( void )`	録音停止
割り込み発生時の処理	`void isr( void )`	録音ハードウェアからのバッファフル通知割り込み時の処理

表2　Sound BlasterのDSP命令の例

時定数（time constant）とは，65536－（256 000 000/（channels * sampling rate））で表される定数のことである．サンプリング周波数やステレオ/モノラルの切り替え時に設定し直す必要がある．

DSPコマンド	内　容
40h	時定数の設定
48h	内部バッファ・サイズの設定
2Ch	録音開始（8ビットauto-init DMAモード）
DAh	録音停止（auto-init DMA終了）

表3　Sound BlasterのI/Oポートの例

I/Oアドレス	内　容	Read/Write
22Ch	DSP Write Command/Data	Write
22Eh	DSP Read Buffer Status（Bit 7）	Read Only

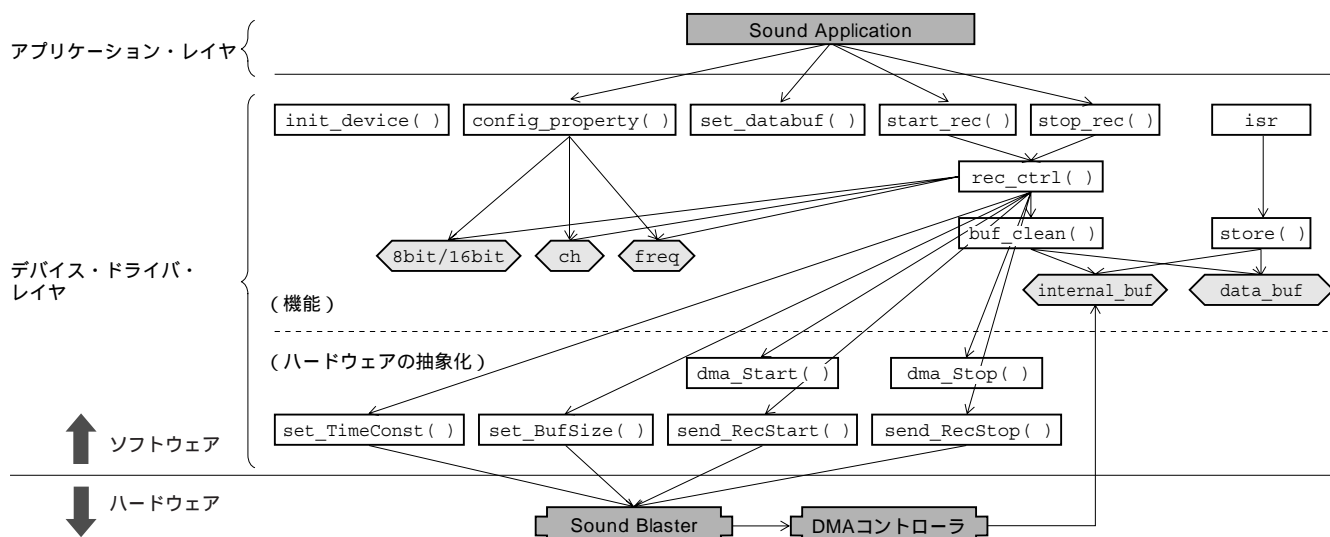



図9 Sound Blasterデバイス・ドライバの構造
ハードウェア依存部と普遍部をレイヤで分けた，レイヤ・アーキテクチャ設計になっている．

リスト1 インターフェースのプログラム例

```
/* Sound Blasterに時定数を設定 */
void set_TimeConst(int ch, int rate)
{
    outp(0x22C, 0x40);
    outp(0x22C, 65536 - (256 000 000 / (ch *
rate)));
}

/* Sound Blasterに内部バッファ・サイズを設定 */
void set_BufSize(int InternalBufSize)
{
    outp(0x22C, 0x48);
    outp(0x22C, ((InternalBufSize - 1) & 0xff);
    outp(0x22C, ((InternalBufSize - 1) >> 8);
}

/* Sound Blasterに録音開始を指示 */
void send_RecStart(void)
{
    outp(0x22C, 0x2c);
}

/* Sound Blasterに録音終了を指示 */
void send_RecStop(void)
{
    outp(0x22C, 0xda);
}
```

します．上位に位置するモジュールが目的達成のために下位モジュールを呼び出す階層構造を，ぜひ参考にしてみてください．また，DSP命令の表(**表2**)を基に作成したインターフェースのプログラム例を，**リスト1**に示します．

● プラットホームとつなげる

デバイス・ドライバの構造を設計できたら，最後にプラットホームと接続する方法を設計します．

プラットホームによって，スレッドやハードウェアとの協調動作の実現方法が変化します．つまり，「割り込みは何本あってタイマは何本使えるか」といったマイコンの仕様や，リアルタイムOSや高級OSが搭載されているかどうかで，スレッドやハードウェアとの協調動作の実現方法が決まるのです．よって，まずはこの辺りの仕様を把握します．ここでも，ハードウェアやOS，マイコンのマニュアルを読むことになります．繰り返しますが，地道ながら大切な作業です．

マイコンの仕様やOSの仕様をつかみ，スレッドや協調動作の実現方法がイメージできたら，デバイス・ドライバの構造図と合わせて接続方法を設計します．Sound Blasterのデバイス・ドライバをmain関数と割り込みから呼び出す形で使用する場合の全体構造を**図10**に示します．

● 設計は自由で高度な知的作業

最近，お笑い芸人で，腕を上に突き上げて「自由だあああ」と叫んでいる人がいます．あれはいいですね．ソフトウェアの設計も，まずは自由に考えればよいのです．設計に唯一の正解はないのです．それに，新しい発想は自由から生まれます．

今回はデバイス・ドライバの設計を題材に，組み込みソフトウェアの設計を解説してきました．デバイス・ドライバの設計を行うためには，かなり地道な分析と，トップダウンの視点による情報整理が必要です．そして，Sound Blasterのようなアイデアは，地道な分析の裏付けの上に，

自由な発想を加えて初めて生まれるものなのです．一見相反するように見える「地道」と「自由」の両方ができてこそ初めて，優れたエンジニアになれると思います．

これからソフトウェア・エンジニアとしての道を歩んでいく皆さんも，まずは地道に本質を追究し，きちんとものづくりができるようになってください．そして，もっと上を目指すために，自由な発想により新しいことを生み出し，無駄のない「機能美」を作り出せるような，本物のエンジニアを目指して頑張ってください．

図10　システム・プラットホームとの統合
main関数で始まるアプリケーション・ソフトウェアの例を示す．

参考・引用*文献

(1) 茂木 健一郎；ニュートンの「アハ！」体験，日本経済新聞，2005年8月25日夕刊，http://kenmogi.cocolog-nifty.com/qualia/2005/08/post_5513.html
(2) Creative Labs；Sound Blaster Hardware Programming Guide, http://www.nondot.org/sabre/os/files/Sound/SoundBlaster/sb-hardware-prog-guide.pdf

もり・たかお
三栄ハイテックス(株) ソフト開発部

<筆者プロフィール>
森 孝夫．組み込みシステムやソフトウェアの開発関連のコンサルティング，火消しを仕事としている．高校で数学の非常勤講師をして以来，教育に関する問題を考えるのがライフ・ワーク．現在は，技術者倫理と設計時の思考に着目した組み込み開発の教育カリキュラムを検討中．イタリア料理，韓国ドラマ，「花より男子」をこよなく愛する．最近のマイブームは「嵐」の「Love so sweet」をカラオケで歌うこと．

3